Comp160
Lab 3
Spring 2019

For this lab we'll work on a series of functions that would prove useful for developing a bouncing-ball animation program like you saw on the first day of class. The emphasis is basic function development with regular, incremental testing.

Lab 4

Just like last week, each function you define should include with it a brief description of the function's purpose as well as several concrete examples. When you use conditionals, work on developing examples that cover all the cases of the conditional. DrRacket will show you cases not covered if you run all your examples at once. Code that is not used is highlighted after you run examples. To make running examples as tests a bit easier, try adopting the following style[1]:

[1] If you're feeling pretty good about functions then feel free to work on formal testing as seen in Section 3.5 of the text.

```
1
2 ; Exercise 192
3
4 ; This function computes ... given ...
5 (define (foo x y z)
6   ...)
7
8 ; (foo 1 2 3) ; --> 14
9 ; (foo 4 5 6) ; --> 21
```

Notice the expected answer is commented out within the line such that delete the first semicolon leaves the function call uncommented but the expected value still a comment. You can now uncomment your examples for the function you're working on and easily run them to see if your on track. Alternatively, run the expression in the interactions window and compare the result to what you expected. Either way, the goal here is to get some confirmation from the computer that your function is working as expected.

The Bouncy Ball Program

Imagine a program that animates a ball bouncing freely around a rectangular space. We'll use x and y to represent the ball's current location in the space[2] and dx and dy to represent the distance it moves in the x and y directions, respectively, in a single frame of animation.

[2] Remember (0,0) is the upper left hand corner

See it Move

To the computer the current *state* of the ball is simply the value of the four variables: x, y, dx, and dy. We tend to think in terms of the picture implied by those four values. So having a way to visualize the current location of the ball will help us as we develop this program. Lets start by working out some code to do just that.

1. Write a function named *draw-ball* that takes x and y, the location of the ball, and computes an image with a ball at those coordinates within the space. The space should be 500 pixels wide and 300 pixels tall. The ball should have a diameter of 50.

Now that we can literally see what's happening, let's look at how the ball moves. A no-frills first crack at getting the ball movement functions working might look like this:

```
;; compute the next x coordinate for the ball
(define (ball-next-x x dx)
  (+ x dx))

;; compute the next y coordinate for the ball
(define (ball-next-y y dy)
  (+ y dy))
```

Copy the definitions for ball-next-x and ball-next-y to your definitions window. We know from draw-ball that the location of the ball is always comprised of positive numbers. Movement, however, must be either positive or negative in order for the ball to move freely in the allotted space. Just like we think of the location of the ball as the pair (x,y), we think of the movement of the ball as the pair (dx,dy). The sign of the variables dx and dy determines the general direction of the ball. There are four combinations of signs for dx and dy: both positive, both negative, positive dx with negative dy, and negative dx with positive dy. Let's make sure we understand how to interpret these combos.

2. Come up with at least one concrete example of a (dx,dy) pair for each of four sign combinations listed above. In your examples, restrict the value of dx and dy to have an absolute value less than 25 but greater than 0. In comments, write out each pair and indicate which direction the ball is moving.

3. Test your understanding of the directions associated with the different (dx,dy) pairs by writing up concrete examples that give us a clear before and after picture of a single step of the animation. Start with the ball somewhere in the middle of the space. Now write up a few expression of the following form:

```
(draw-ball X Y)
(draw-ball (ball-next-x X DX) (ball-next-y Y DY))
```

In the place of X, DX, Y, and DY in your expressions should be the literal values you've chosen. Running these examples will give you a before and after image. Run the examples with the stepper as well to get a feel for how it works. Do as many examples as you need to in order to feel like you have some basic intuition for how (x,y) and (dx,dy) translates into ball movement.

We can now draw the ball and, to some extent, move the ball. Let's turn our attention to bouncing the ball off a wall.

Bouncing Around the Room

The ball bounces when it makes contact with one of the walls. This means our program needs a function that tells us if the ball is currently in contact with the wall. We'll break this into two problems, has it hit the left or right walls or has it hit the top or bottom.

4. A ball has "hit the wall" if some part of it is at or beyond the wall when drawn. Assume that no ball will ever go half way or more through a wall. For each wall, come up with at least one concrete (x,y) pair for a ball that has "hit that wall". Write BSL expressions to draw each of your examples showing the ball at or beyond the wall.

5. Write a function named has-hit-x? that when given the balls current x location will compute the boolean true if the ball hit the left or right wall and false otherwise. Because the x location of a ball is the center of the ball, you'll need to account for the fact that the edge of the ball with radius r is at most $x \pm r$.

6. Write a function named has-hit-y? that checks if the ball has hit the top or bottom of the area.

Now that we can determine when a ball has struck the wall, we can think about what happens when it bounces. A bounce is a change in direction. When the ball hits the left or right wall, then the sign of dx changes, positive to negative or negative to positive. When it hits the top or bottom wall, then the sign of dy changes as well.

7. Write the function ball-next-dx that computes the dx value for the next animation frame given the current x and dx values. If no bounce occurred on the right or left wall, then the dx value stays as is. If a bounce occurs[3] then the sign on the dx should change.

[3] the ball hits the wall

8. Write the function ball-next-dy that computes the dy value for the next animation frame given the current y and dy values. If no bounce occurred on the top or bottom wall, then the dy value stays as is. If a bounce occurs then the sign on the dy should change.

Drawing Revisited

Wouldn't it be cool if the ball changed color when it bounced? Let's do it! Go back to draw-ball and change the code so that the ball changes color if it's going to bounce, i.e. if the ball is currently hitting the wall, then draw it a different color than usual.

9. Rewrite draw-ball so that the color is different if the ball has in contact with a wall. Your previously developed concrete examples for hitting the wall should help you test if your new draw-ball is fully functioning. If not, then develop additional concrete examples to be sure your ball is changing color when it should.

Go Faster

DrRacket translates keyboard presses into strings to make it easy for you to build programs reacting to user input. A complete list of key event strings can be found in the documentation[4]. We'll only concern ourselves with the arrow keys: "up" and "down".

[4] See Key Events

The up and down arrows should speed up and slow down the ball. This translates into increasing dx and dy by 1 relative to it's current sign. If it's negative, then add negative one rather than positive one. To avoid problems we need to restrict the speed of the ball such that the absolute values of dx and dy are each strictly greater than zero but also strictly less than the radius of the ball. If either is at the current maximum or minimum, then pressing up or down should have no effect.

10. Write a pair of functions named on-key-dx and on-key-dy that take a key event string as well as dx and dy, respectively, and compute the new value of dx/dy given the key pressed. If any key besides up or down arrow is pressed, then your function should compute the current value of dx/dy.

You're Done

If you've made it this far then you're done with the lab. Print your definitions and hand them in. In class we'll talk about what's missing in order for us to get DrRacket to animate and run this program for us.